The probability density function (pdf) for a 2-dimensional real-valued random vector $\mathbf{X}$ is as follows: $p(\mathbf{x}) = P(L=0)p(\mathbf{x}|L=0) + P(L=1)p(\mathbf{x}|L=1)$. Here L is the true class label that indicates which class-label-conditioned pdf generates the data.

The class priors are $P(L=0) = 0.6$ and $P(L=1) = 0.4$. The class class-conditional pdfs are $p(\mathbf{x}|L=0) = w_{01}g(\mathbf{x}|\mathbf{m}_{01},\mathbf{C}_{01}) + w_{02}g(\mathbf{x}|\mathbf{m}_{02},\mathbf{C}_{02})$ and $p(\mathbf{x}|L=1) = w_{11}g(\mathbf{x}|\mathbf{m}_{11},\mathbf{C}_{11}) + w_{12}g(\mathbf{x}|\mathbf{m}_{12},\mathbf{C}_{12})$, where $g(\mathbf{x}|\mathbf{m},\mathbf{C})$ is a multivariate Gaussian probability density function with mean vector $\mathbf{m}$ and covariance matrix $\mathbf{C}$. The parameters of the class-conditional Gaussian pdfs are: $w_{i1} = w_{i2} = 1/2$ for $i \in \{1,2\}$, and

$$\mathbf{m}_{01} = \begin{bmatrix}-1\\-1\end{bmatrix} \quad \mathbf{m}_{02} = \begin{bmatrix}1\\1\end{bmatrix} \quad \mathbf{m}_{11} = \begin{bmatrix}-1\\1\end{bmatrix} \quad \mathbf{m}_{12} = \begin{bmatrix}1\\-1\end{bmatrix} \quad \mathbf{C}_{ij} = \begin{bmatrix}1&0\\0&1\end{bmatrix} \text{ for all } \{ij\} \text{ pairs.}$$

For numerical results requested below, generate the following independent datasets each consisting of iid samples from the specified data distribution, and in each dataset make sure to include the true class label for each sample.

- D^{20}_{train} consists of 20 samples and their labels for training;
- D^{200}_{train} consists of 200 samples and their labels for training;
- D^{2000}_{train} consists of 2000 samples and their labels for training;
- $D^{10K}_{validate}$ consists of 10000 samples and their labels for validation;

Part 1: (6%) Determine the theoretically optimal classifier that achieves minimum probability of error using the knowledge of the true pdf. Specify the classifier mathematically and implement it; then apply it to all samples in $D^{10K}_{validate}$. From the decision results and true labels for this validation set, estimate and plot the ROC curve for a corresponding discriminant score for this classifier, and on the ROC curve indicate, with a special marker, the location of the min-P(error) classifier. Also report an estimate of the min-P(error) achievable, based on counts of decision-truth label pairs on $D^{10K}_{validate}$. Optional: As supplementary visualization, generate a plot of the decision boundary of this classification rule overlaid on the validation dataset. This establishes an aspirational performance level on this data for the following approximations.

Part 2: (12%) (a) Using the maximum likelihood parameter estimation technique train three separate logistic-linear-function-based approximations of class label posterior functions given a sample. For each approximation use one of the three training datasets D^{20}_{train}, D^{200}_{train}, D^{2000}_{train}. When optimizing the parameters, specify the optimization problem as minimization of the negative-log-likelihood of the training dataset, and use your favorite numerical optimization approach, such as gradient descent or Matlab's fminsearch. Determine how to use these class-label-posterior approximations to classify a sample in order to approximate the minimum-P(error) classification rule; apply these three approximations of the class label posterior function on samples in $D^{10K}_{validate}$, and estimate the probability of error that these three classification rules will attain (using counts of decisions on the validation set). Optional: As supplementary visualization, generate plots of the decision boundaries of these trained classifiers superimposed on their respective training datasets and the validation dataset. (b) Repeat the process described in Part (2a) using a logistic-quadratic-function-based approximation of class label posterior functions given a sample.

Discussion: (2%) How does the performance of your classifiers trained in this part compare to each other considering differences in number of training samples and function form? How do they compare to the theoretically optimal classifier from Part 1? Briefly discuss results and insights.

Note 1: With $\mathbf{x}$ representing the input sample vector and $\mathbf{w}$ denoting the model parameter vector, logistic-linear-function refers to $h(\mathbf{x},\mathbf{w}) = 1/(1+e^{-\mathbf{w}^{\mathrm{T}}\mathbf{z}(\mathbf{x})})$, where $\mathbf{z}(\mathbf{x}) = [1, \mathbf{x}^{\mathrm{T}}]^{\mathrm{T}}$; and logistic-quadratic-function refers to $h(\mathbf{x},\mathbf{w}) = 1/(1+e^{-\mathbf{w}^{\mathrm{T}}\mathbf{z}(\mathbf{x})})$, where $\mathbf{z}(\mathbf{x}) = [1, x_1, x_2, x_1^2, x_1x_2, x_2^2]^{\mathrm{T}}$.

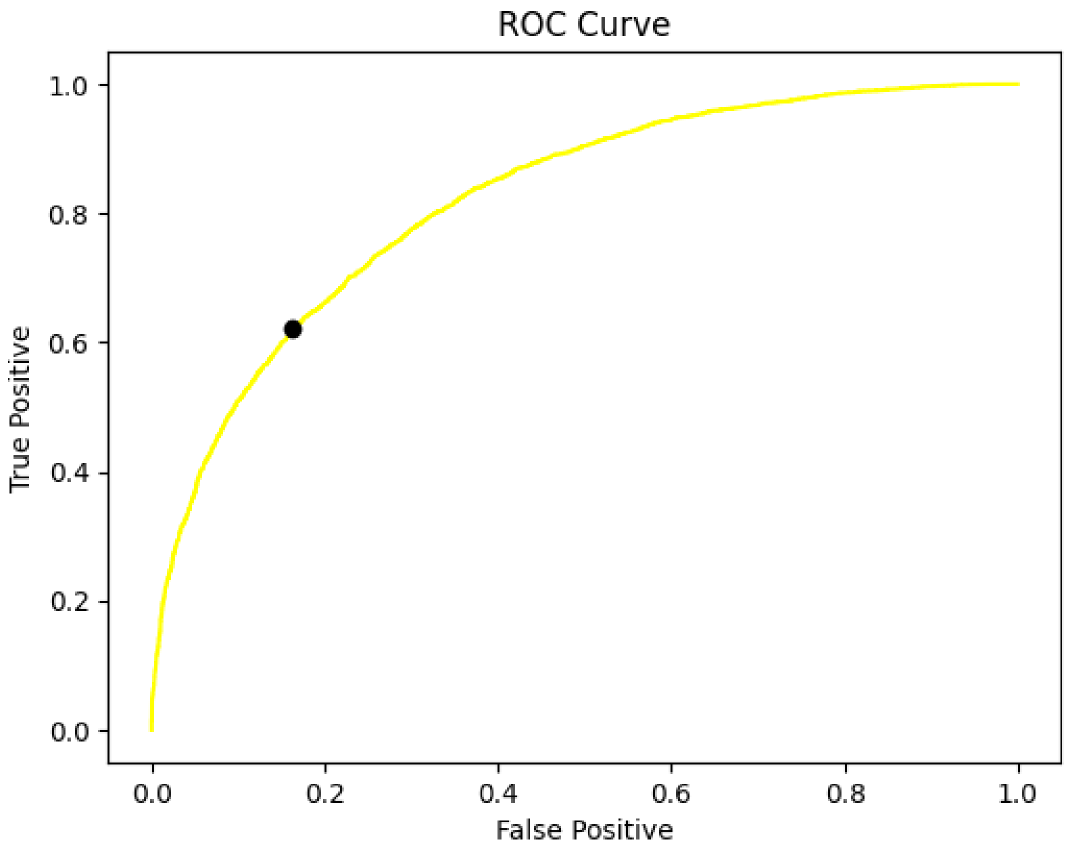
Theoretical Optimal Classification Classification Rule:

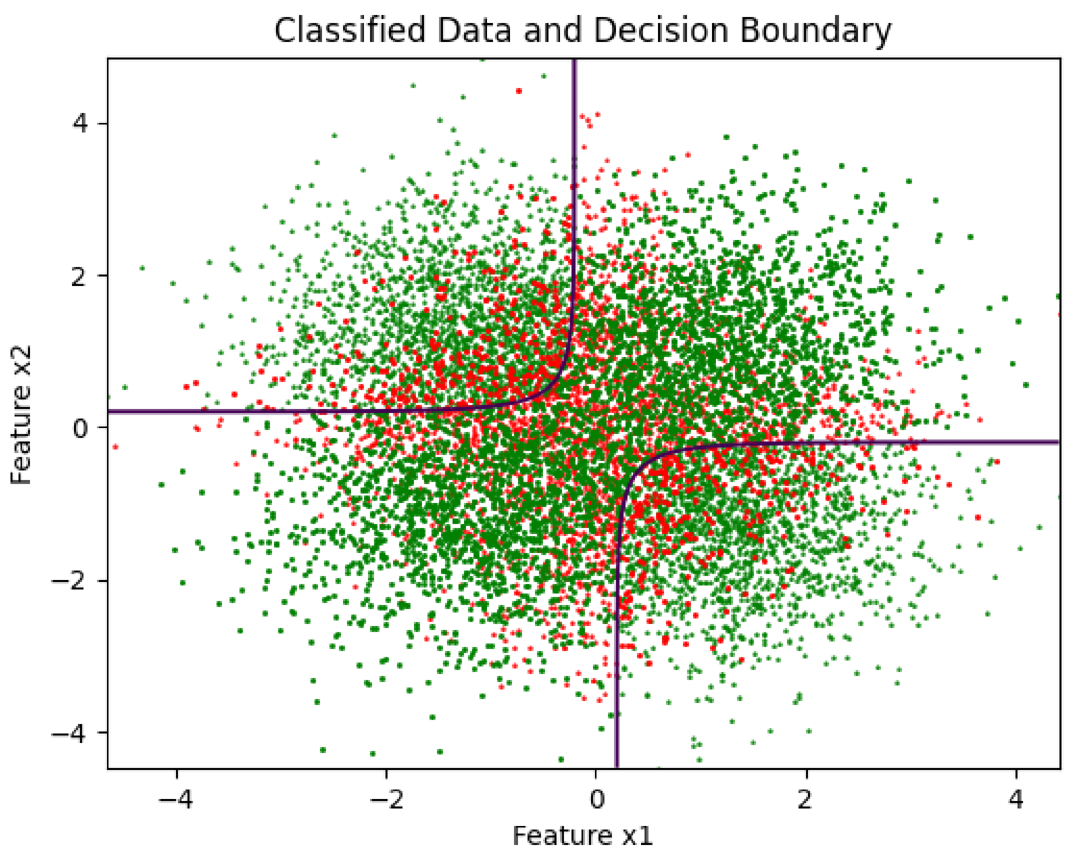
$$\frac{P(X|L = 1)}{P(X|L = 0)} \underset{(D = 0)}{\overset{(D = 1)}{\gtrless}} \frac{(6)(\lambda10 - \lambda00)}{(4)(\lambda01 - \lambda11)}$$

The loss function chosen was 0-1 loss with 1 point penalty for all incorrect decisions. Thus, $\lambda10 = \lambda01 = 1$ and $\lambda00 = \lambda11 = 0$

$$\frac{P(X|L = 1)}{P(X|L = 0)} \underset{(D = 0)}{\overset{(D = 1)}{\gtrless}} \frac{(6)}{(4)}$$

The classifier was trained using the above rule and tested on the 10000 validate samples. ROC curve with the highlighted minimum probability of error is shown below.





Value of Gamma (Ideal) is **1.5**
Corresponding minimum error is **0.25115**
Value of Gamma (practical) is **1.42845**
Corresponding minimum error is **0.24921**

* Linear logistic Regression:→

I/p Vector = $[x_1, x_2]$

Basis Vector = $[1 \; x^T]^T$

O/p function $h(x, W)$ = $\frac{1}{1+e^{-W^T b(x)}} = \begin{cases} 1 \; ; \text{ if } h(x) \geq 0.5 \\ 0 \; ; \text{ if } h(x) < 0.5 \end{cases}$

* All values in the weight vector W are intilized to 0. & the optimization problem is solved by using "Nelder method. -Mead"

$$W_{ML} = \text{argmin} \; -\ln P(L|x, w)$$

$$\ln(P(L|x,w)) = \begin{cases} \ln h(x,W) & ; \text{if label} = 1 \\ 1 - \ln h(x,W) & ; \text{if label} = 0 \end{cases}$$

* Combining both Equations we get.

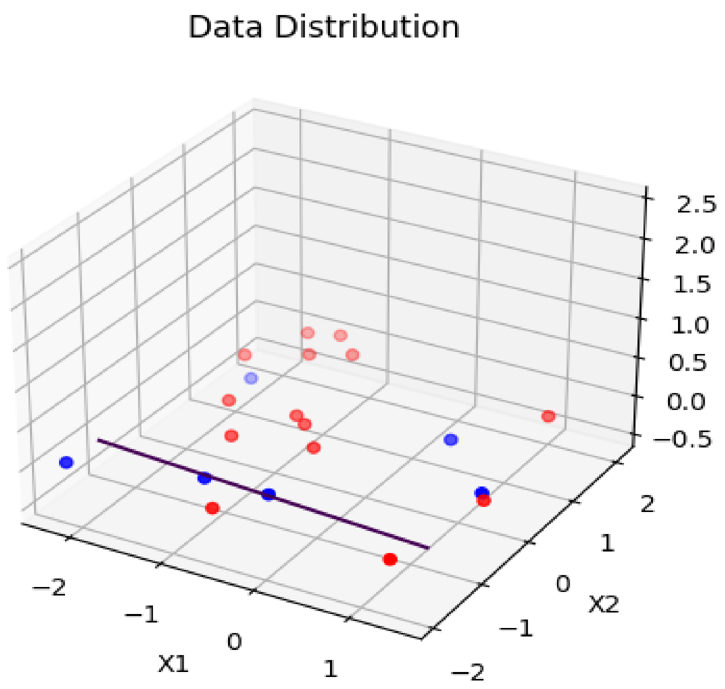
$$W_{ML} = \underset{W}{\text{argmin}} - \left[L \cdot h(x,w) + (1-L) \cdot \ln(1-h(x,w))\right]$$

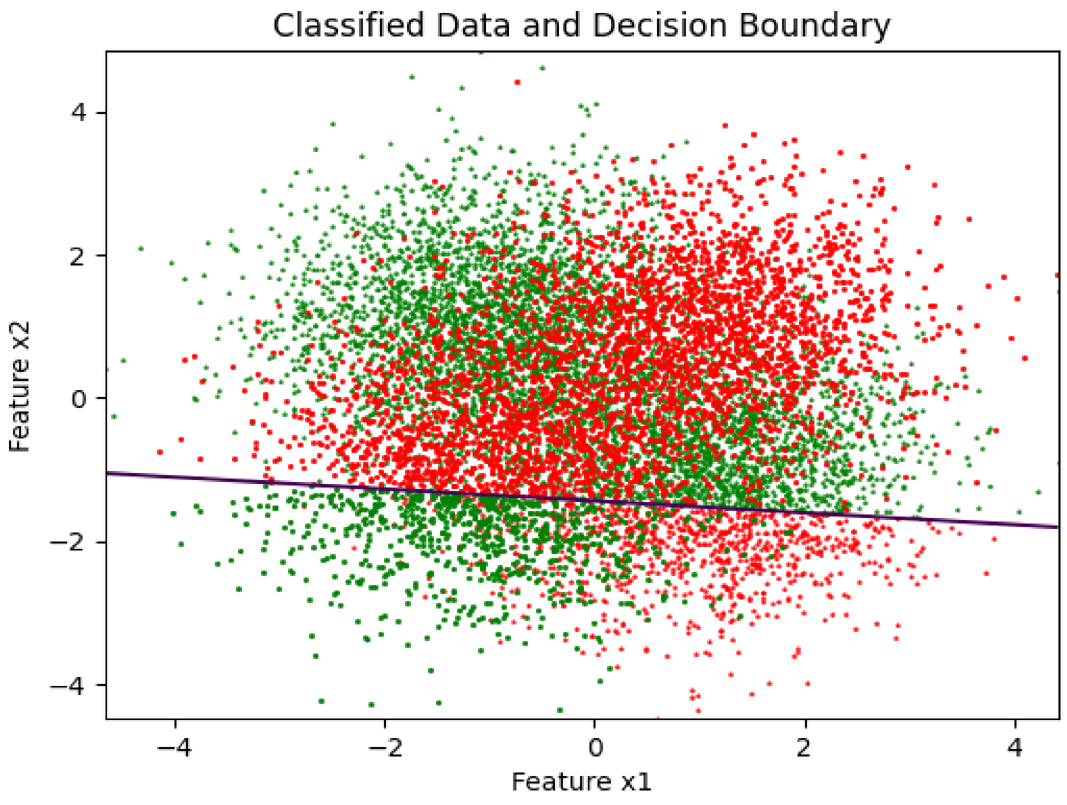
* Quadratic logistic Regression:→

Basis vector = $[1, x_1, x_2, x_1^2, x_1x_2, x_2^2]$

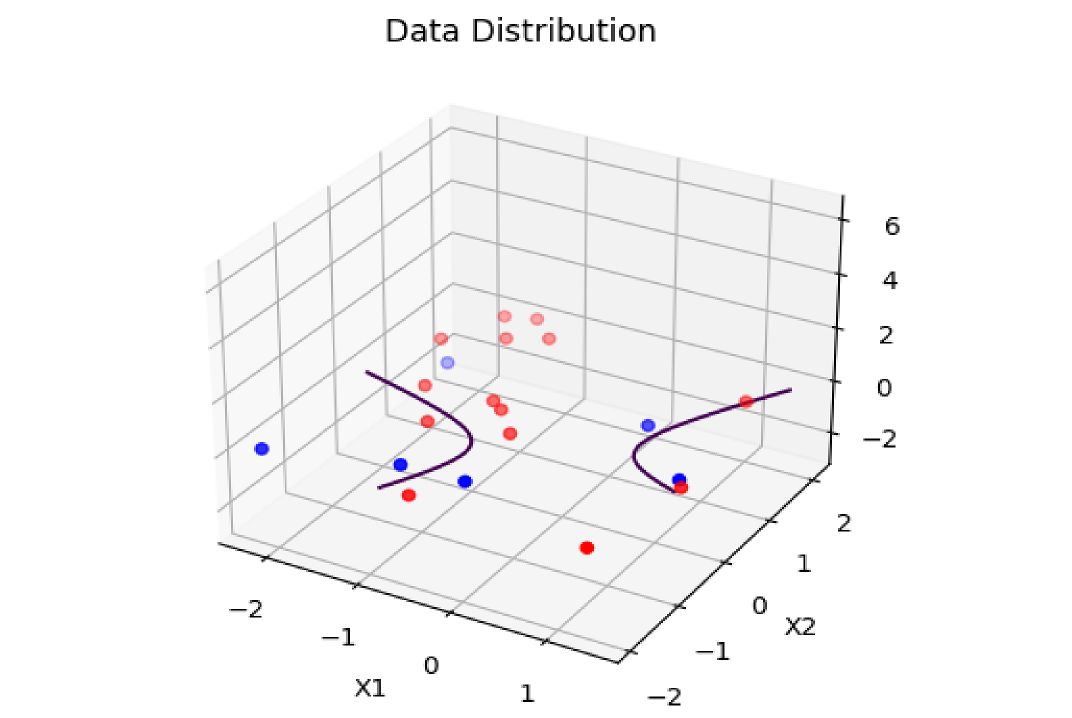
Model training procedure & Cost function are same as the above.

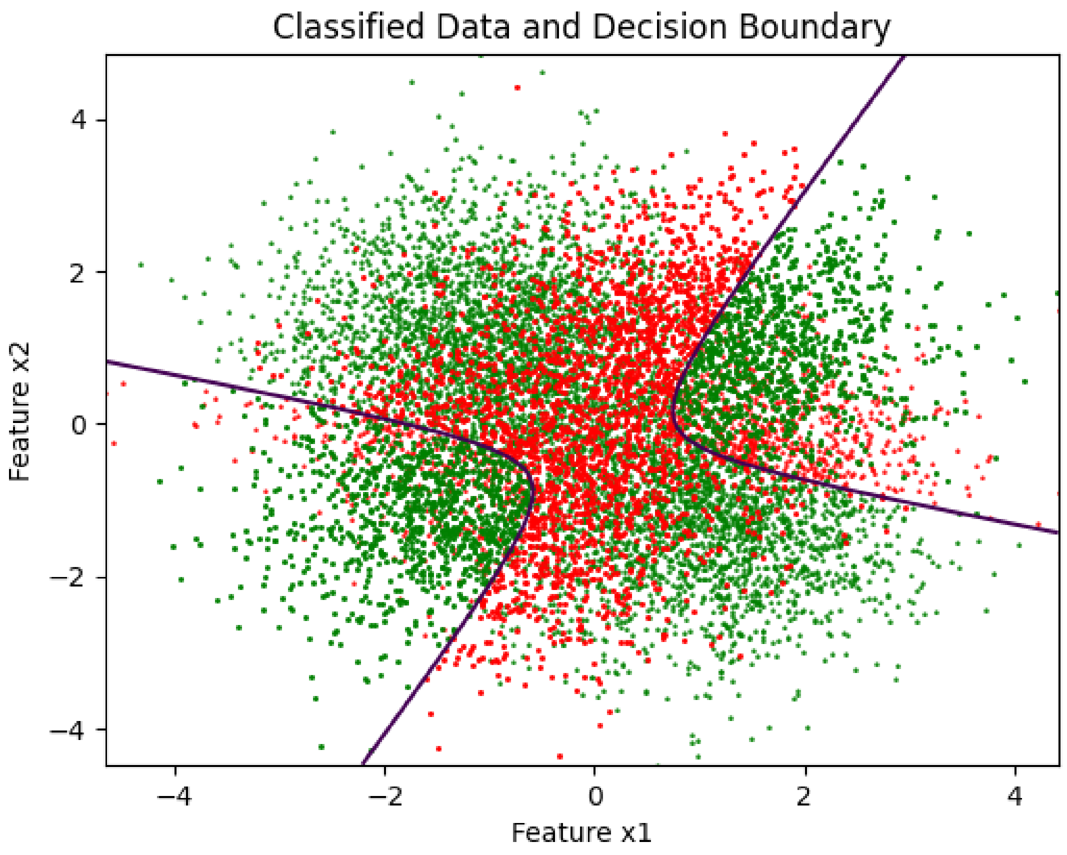
Logistic Linear Regression (Dtrain = 20)



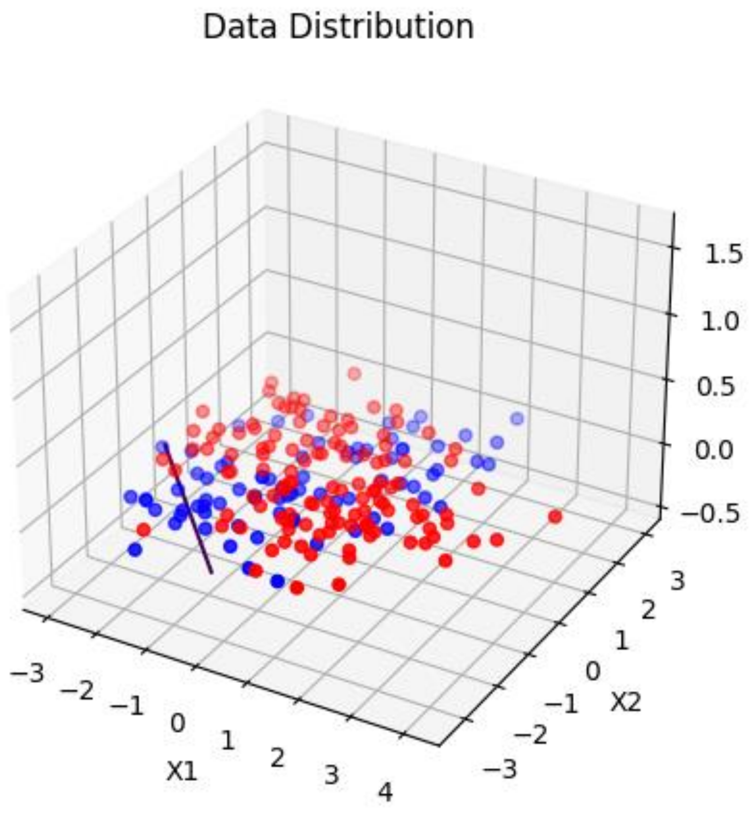


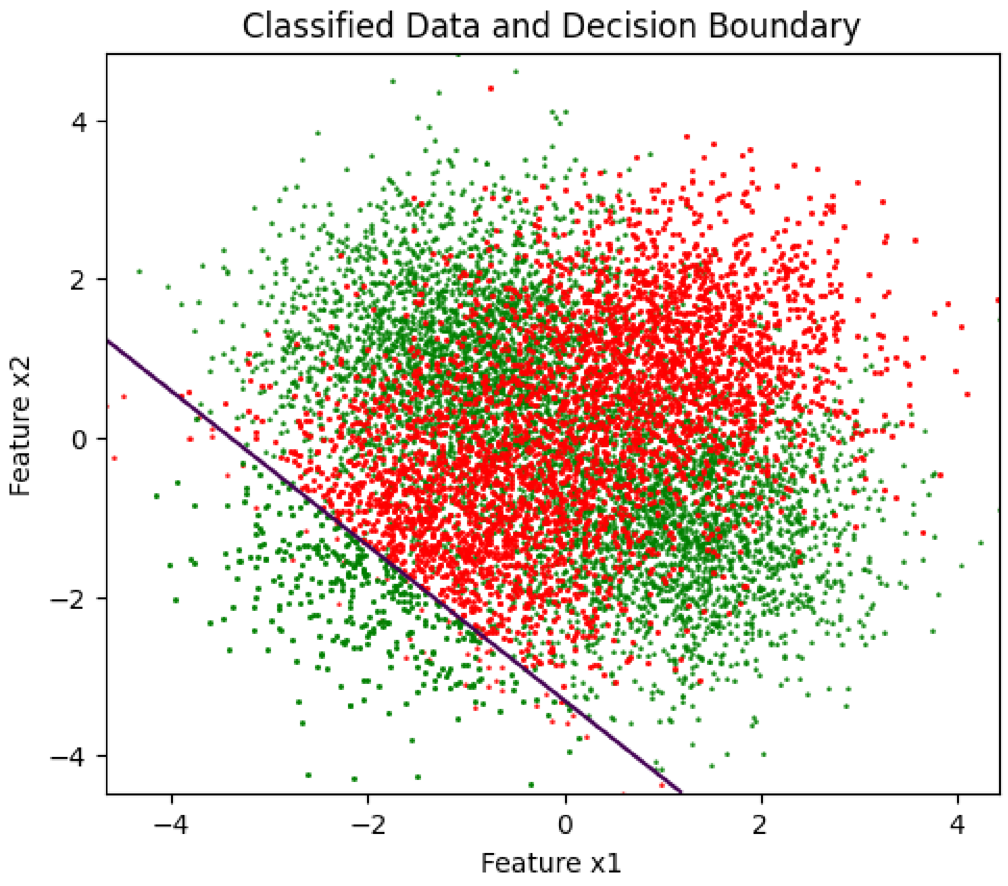
Logistic Quadratic Regression (Dtrain = 20)



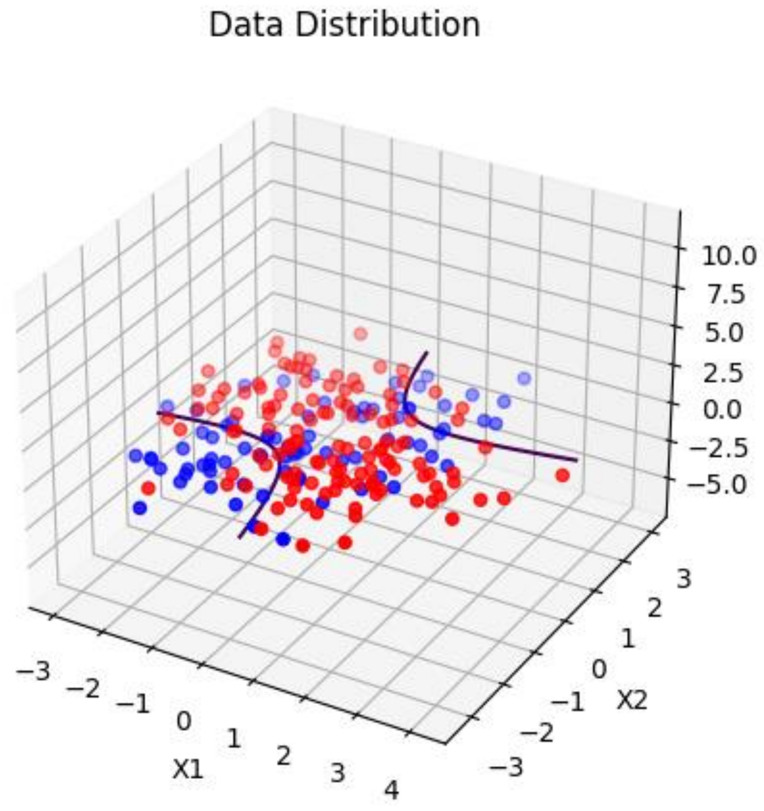


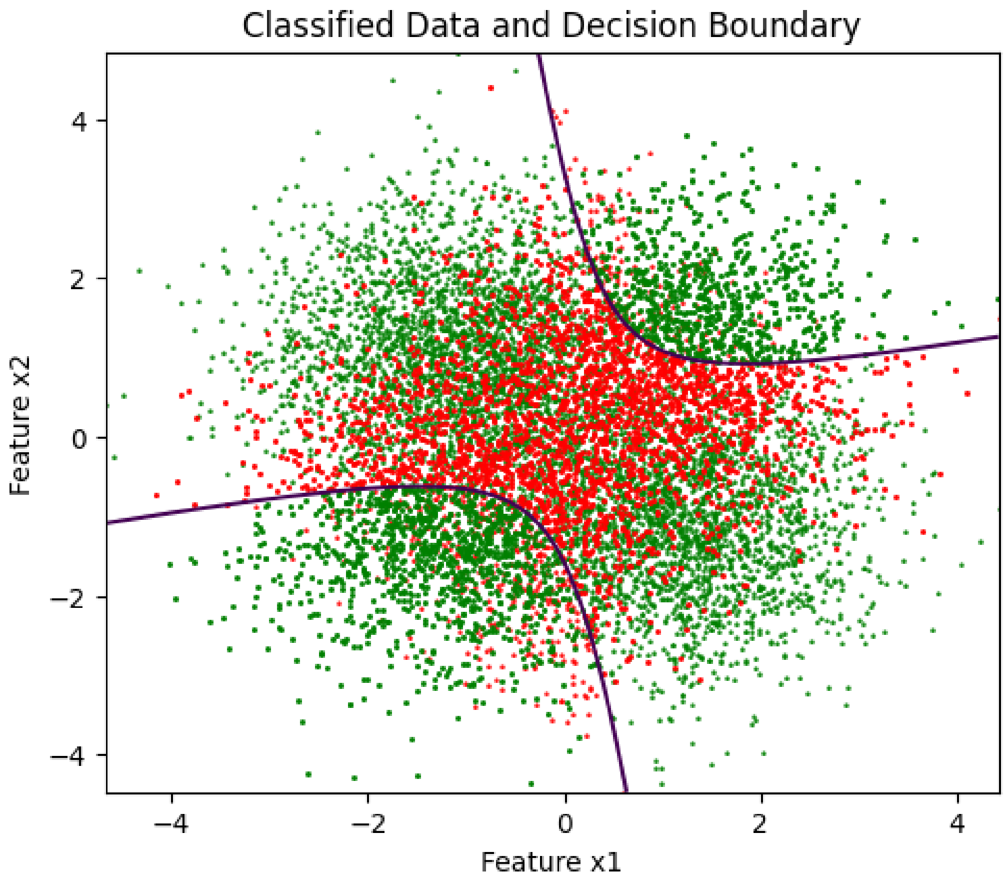
Logistic Linear Regression (Dtrain = 200)



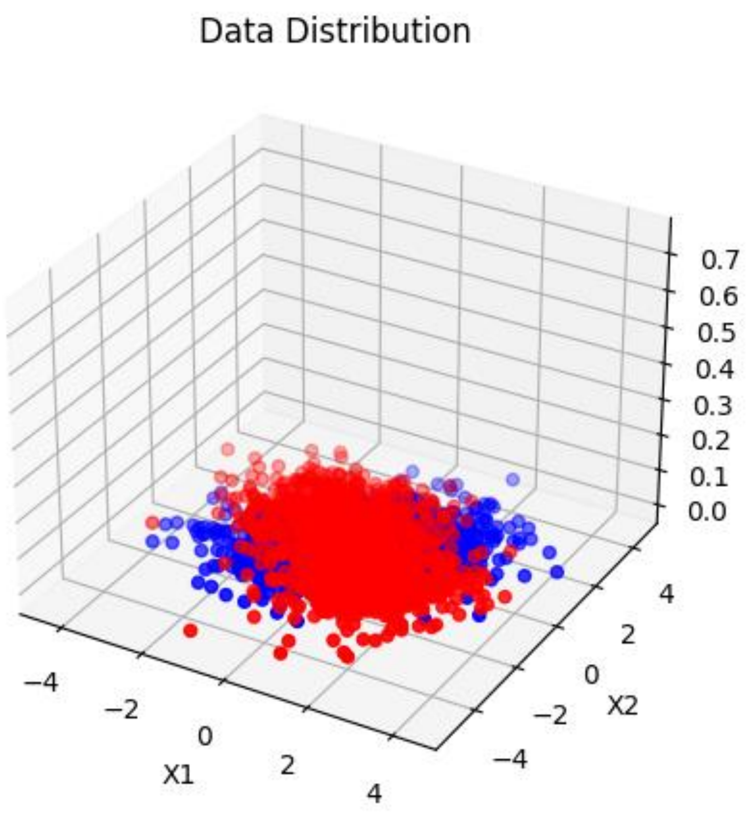


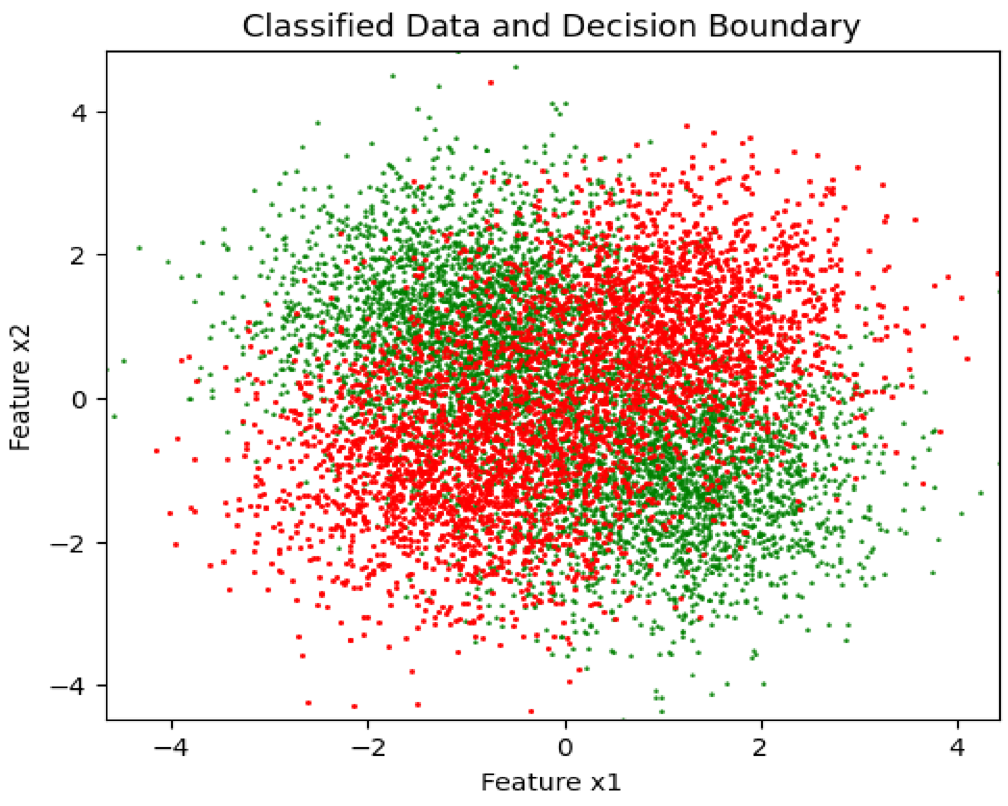
Logistic Quadratic Regression (Dtrain = 200)



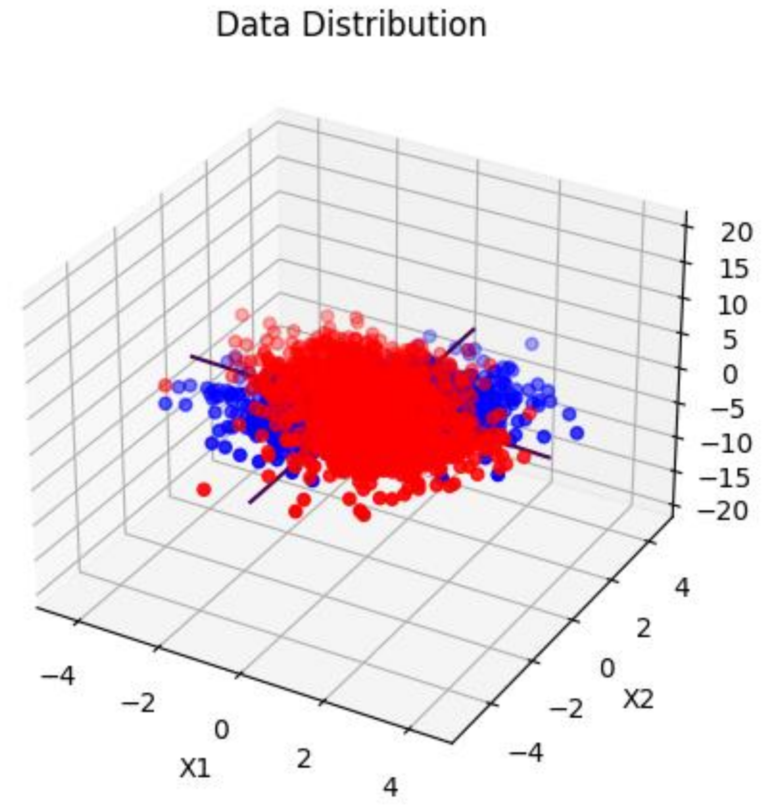


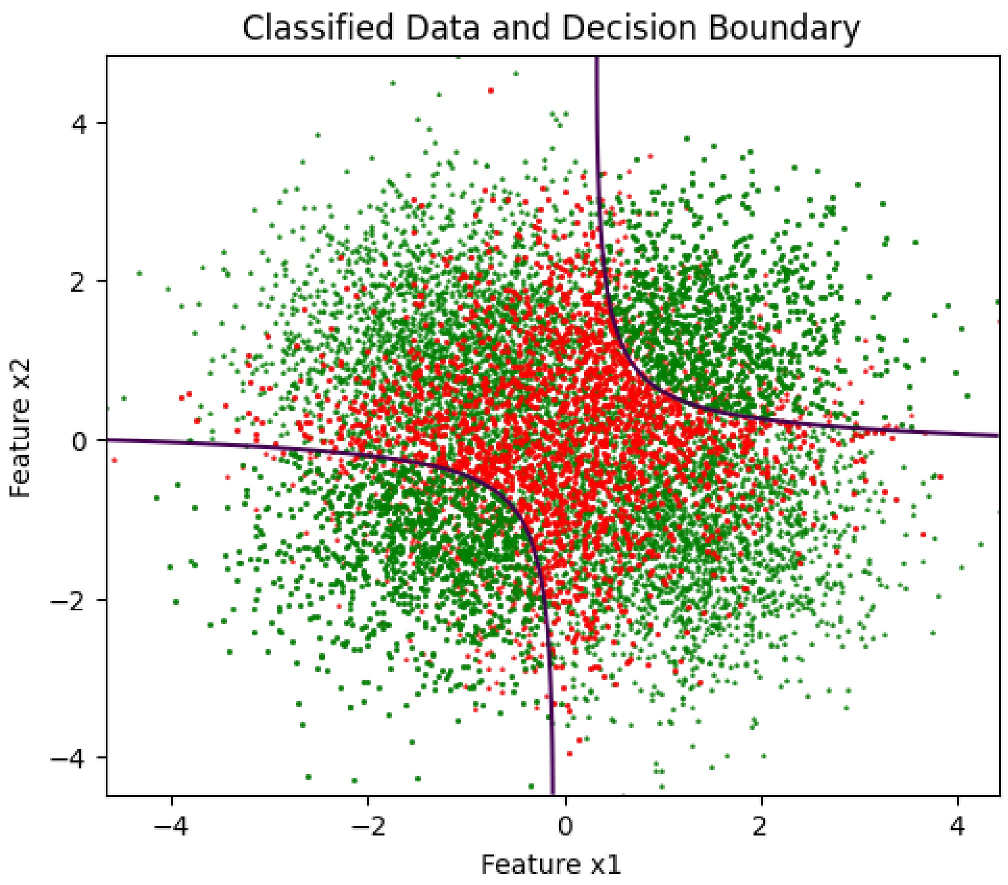
Logistic Linear Regression (Dtrain = 2000)





Logistic Quadratic Regression (Dtrain = 2000)





Error for Logistic Linear Model with 20 samples is 42.96.
Error for Logistic Quadratic Model with 20 samples is 32.2.

Error for Logistic Linear Model with 200 samples is 38.5.
Error for Logistic Quadratic Model with 200 samples is 28.91.

Error for Logistic Linear Model with 2000 samples is 41.19.
Error for Logistic Quadratic Model with 2000 samples is 26.24.

	ERROR Dtrain = 20	**ERROR Dtrain = 200**	**ERROR Dtrain = 2000**
Logistic Linear Model	**42.96**	**38.5**	**41.19**
Logistic Quadratic Model	**32.2**	**28.91**	**26.24**

Observations:

- The Logistic-Quadratic Model performs better than the Logistic-Linear Model based on visual and numerical results.
- The quadratic model considers the relationship between two features during decision-making.
- The classes in the data distribution are not linearly separable and an elliptical partition is effective in defining the boundary.
- With more training samples, the model can identify suitable parameters and construct a better decision boundary, leading to a reduction in error rate.
- With larger training samples, the error rate of the Logistic-Quadratic Model is similar to that of the ideal Bayesian Classifier.

Appendix:

```
import numpy as np
import matplotlib as plt
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from scipy.optimize import minimize
from mpl_toolkits.mplot3d import Axes3D
np.random.seed(10)

# ALL THE BELOW VARIABLES ARE GLOBAL

# Number of samples
validation_sample_size = 10000  # FOR VALIDATION DATA SET
No_features = 2
No_labels = 2
No_mixtures = 4

# Mean
m01 = [-1, -1]
```

```
m02 = [1, 1]
m11 = [-1, 1]
m12 = [1, -1]

# Covariance
c01 = [[1, 0],[0, 1]]
c02 = [[1, 0],[0, 1]]
c11 = [[1, 0],[0, 1]]
c12 = [[1, 0],[0, 1]]

# class priors
p = [0.6, 0.4]
w = [0.5, 0.5, 0.5, 0.5]

def generate_class_labels(sample_size):
    class_labels = np.random.rand(sample_size)
    class_labels = class_labels >= p[1]
    return class_labels.astype(int)

def generate_samples(sample_size, class_labels):
    samples = np.zeros(shape = [sample_size, No_features])
    for index in range(sample_size):
        if class_labels[index] == 0:
            if (np.random.rand(1,1) >= w[1]):
                samples[index, :] = np.random.multivariate_normal(m01, c01)
            else:
                samples[index, :] = np.random.multivariate_normal(m02, c02)
        elif class_labels[index] == 1:
            if (np.random.rand(1,1) >= w[3]):
                samples[index, :] = np.random.multivariate_normal(m11, c11)
            else:
                 samples[index, :] = np.random.multivariate_normal(m12, c12)
    return samples


def discriminant_score(samples):
    gaussian_pdf_0 = np.log(w[0] * (multivariate_normal.pdf(samples, m01, c01)) + w[1] *
(multivariate_normal.pdf(samples, m02, c02)))
    gaussian_pdf_1 = np.log(w[2] * (multivariate_normal.pdf(samples, m11, c11)) + w[3] *
(multivariate_normal.pdf(samples, m12, c12)))
    return gaussian_pdf_1 - gaussian_pdf_0

def calculate_mid_points(d_score_sorted):
    threshold = []
    for i in range( len(d_score_sorted) - 1 ):
        threshold.append( (d_score_sorted[i] + d_score_sorted[i+1]) / 2.0 )
    threshold = np.array(threshold)
    return threshold

def classify(d_score, threshold, class_labels):
    true_positive = [0] * len(threshold)
    false_positive = [0] * len(threshold)
    error = [0] * len(threshold)
    for (index, th) in enumerate(threshold):
        decision = (d_score >= th)
        true_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 1))))
        true_positive[index] = true_positive[index] / np.size(np.where(class_labels == 1))
        false_positive[index] = (np.size(np.where((decision == 1) & (class_labels == 0))))
        false_positive[index] =  false_positive[index] / np.size(np.where(class_labels == 0))
        error[index] = (p[0] * false_positive[index]) + (p[1] * (1 - true_positive[index]))
    return true_positive, false_positive, error

def cost_function(w, z, train_sample_size, class_labels):
    prediction = 1 / (1 + np.exp(-(np.dot(w.T, z))))
    cost = (-1 / train_sample_size) * ((np.sum(class_labels * np.log(prediction))) + (np.sum((1 -
class_labels) * np.log(1 - prediction))))
    return cost

# Function to decide decision boundary based on classifier type
def decision_boundary(X, w, type):
    hgrid = np.linspace(min(X[:,0]), max(X[:,0]), 100)
    vgrid = np.linspace(min(X[:,1]), max(X[:,1]), 100)
    x1, x2 = np.meshgrid(hgrid,vgrid)
    b = np.zeros((100,100))
    log_gamma = np.log(p[0] / p[1])
    for i in range(len(x1)):
        for j in range(len(x2)):
            if (type == 'T'):
                gaussian_pdf_0 = np.log((w[0] * (multivariate_normal.pdf(np.array([x1[i][j], x2[i][j]]),
m01, c01))) + (w[1] * (multivariate_normal.pdf(np.array([x1[i][j], x2[i][j]]), m02, c02))))
                gaussian_pdf_1 = np.log((w[2] * (multivariate_normal.pdf(np.array([x1[i][j], x2[i][j]]),
m11, c11))) + (w[3] * (multivariate_normal.pdf(np.array([x1[i][j], x2[i][j]]), m12, c12))))
                b[i][j] = gaussian_pdf_1 - gaussian_pdf_0 - log_gamma
            elif (type == 'L'):
                z = np.c_[1, x1[i][j], x2[i][j]].T
                b[i][j] = np.sum(np.dot(w.T, z))
            elif (type == 'Q'):
                z = np.c_[1, x1[i][j], x2[i][j], x1[i][j]**2, x1[i][j] * x2[i][j], x2[i][j] ** 2].T
                b[i][j] = np.sum(np.dot(w.T, z))

    plt.contour(x1, x2, b, levels = [0])
    plt.show()

def plotDistribution(X, y, w, type):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X[np.where(y==0),0],X[np.where(y==0),1],color = 'blue')
    ax.scatter(X[np.where(y==1),0],X[np.where(y==1),1],color = 'red')
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.title('Data Distribution')
    decision_boundary(X, w, type)
    plt.show()

# Function to plot classified data along with decision boundary
def plot_classified_data(X, y, decision, w, type):
    plt.plot(X[np.where((y == 1) & (decision == 1)),0], X[np.where((y == 1) & (decision == 1)),1],
color ='green', marker = '^', markersize = 1)
    plt.plot(X[np.where((y == 1) & (decision == 0)),0], X[np.where((y == 1) & (decision == 0)),1],
color ='red', marker = '^', markersize = 1)
    plt.plot(X[np.where((y == 0) & (decision == 1)),0], X[np.where((y == 0) & (decision == 1)),1],
color ='red', marker = 's', markersize = 1)
    plt.plot(X[np.where((y == 0) & (decision == 0)),0], X[np.where((y == 0) & (decision == 0)),1],
color ='green', marker = 's', markersize = 1)
    plt.xlabel("Feature x1")
    plt.ylabel("Feature x2")
    plt.title("Classified Data and Decision Boundary")
    decision_boundary(X, w, type)

# GENERATING VALIDATION DATA SET AND CLASS LABELS

# generating class labels for the given number of samples
validation_class_labels = generate_class_labels(validation_sample_size)

# generating the samples / data
validation_samples = generate_samples(validation_sample_size, validation_class_labels)

# Calculating discriminant score
validation_set_DS = discriminant_score(validation_samples)
sorted_validation_Ds = np.sort(validation_set_DS)

# calculating the threshold values (mid - points of d_score)
validation_set_th = calculate_mid_points(sorted_validation_Ds)

# classifying the data and calculating the p error
true_positive, false_positive, error = classify(validation_set_DS, validation_set_th,
validation_class_labels)

# computing idea value
ideal_gamma = p[0] / p[1]
ideal_threshold = np.log(ideal_gamma)
ideal_decision = (validation_set_DS >= ideal_threshold)
ideal_true_positive = (np.size(np.where((ideal_decision == 1) & (validation_class_labels ==
1))))/np.size(np.where(validation_class_labels == 1))
ideal_false_positive = (np.size(np.where((ideal_decision == 1) & (validation_class_labels ==
0))))/np.size(np.where(validation_class_labels == 0))
ideal_error = (p[0] * ideal_false_positive) + (p[1] * (1 - ideal_true_positive))
print(f'Value of Gamma (Ideal) is {round(ideal_gamma, 5)} ')
print(f'Corresponding minimum error is {round(ideal_error, 5)}')

# Plot ROC curve
plt.plot(false_positive, true_positive, color = 'yellow')
plt.xlabel('False Positive')
plt.ylabel('True Positive')
plt.title('ROC Curve')
plt.plot(false_positive[np.argmin(error)], true_positive[np.argmin(error)],'o',color = 'black')
plt.show()
print(f'Value of Gamma (practical) is {round(np.exp(validation_set_th[np.argmin(error)]), 5)} ')
print(f'Corresponding minimum error is {round(np.min(error), 5)}')

plot_classified_data(validation_samples, validation_class_labels, ideal_decision, w, 'T')

choice = int(input("Select the Size of the training samples (PRESS 1, 2, or 3) : \n1. 20 \n2. 200 \n3.
2000 \n"))
print(choice)
if (choice == 1):
    train_data_sample_size = 20
elif (choice == 2):
    train_data_sample_size = 200
elif (choice == 3):
    train_data_sample_size = 2000

y = generate_class_labels(train_data_sample_size)
train_data_samples = generate_samples(train_data_sample_size, y)

# LOGISTIC LINEAR MODEL
x_L = np.column_stack((np.ones(train_data_samples.shape[0]), train_data_samples)).T
initial_guess_L = np.zeros(shape = (3, 1)) # similar to variables (a,b,c) initialization
# return information about the optimization process, min value of the function, value of the argument
at which the min occurs
cost_L = minimize(cost_function, initial_guess_L, args = (x_L, train_data_sample_size, y), method =
"Nelder-Mead")
# extracting only the decision variables at the optimal solution (i.e.. where the cost function is
minimum)from the minimize function output
optimal_solution_L = cost_L.x
plotDistribution(train_data_samples, y, optimal_solution_L, 'L')

test_L = np.column_stack((np.ones(validation_samples.shape[0]), validation_samples)).T
# Make decisions to test the models
decision_L = []
decision_L = (1 / (1+ np.exp(-(np.dot(optimal_solution_L.T, test_L)))) >= 0.5).astype(int)
# Evaluate the model and compute error
TP_L = (np.size(np.where((decision_L == 1) & (validation_class_labels == 1))))
TN_L = (np.size(np.where((decision_L == 0) & (validation_class_labels == 0))))
print("Error for Logistic Linear Model with ", train_data_sample_size, " samples is ", (10000 - (TP_L
+ TN_L))/100)
plot_classified_data(validation_samples, validation_class_labels, decision_L, optimal_solution_L, 'L')

# LOGISTIC QUADRATIC MODEL
x_Q = np.column_stack((np.ones(train_data_samples.shape[0]), train_data_samples,
np.square(train_data_samples[:, 0]), np.multiply(train_data_samples[:, 0], train_data_samples[:, 1]),
np.square(train_data_samples[:, 1]))).T
initial_guess_Q = np.zeros(shape = (6, 1))
```

```
# return information about the optimization process, min value of the function, value of the argument
at which the min occurs
cost_Q = minimize(cost_function, initial_guess_Q, args = (x_Q, train_data_sample_size, y), method
= "Nelder-Mead")
# extracting only the decision variables at the optimal solution (i.e.. where the cost function is
minimum)from the minimize function output
optimal_solution_Q = cost_Q.x
plotDistribution(train_data_samples, y, optimal_solution_Q, 'Q')

test_Q = np.column_stack((np.ones(validation_samples.shape[0]), validation_samples[:, 0],
validation_samples[:, 1], np.square(validation_samples[:, 0]), np.multiply(validation_samples[:, 0],
validation_samples[:, 1]), np.square(validation_samples[:, 1]))).T
decision_Q = []
decision_Q = (1 / (1+ np.exp(-(np.dot(optimal_solution_Q.T, test_Q)))) >= 0.5).astype(int)
TP_Q = (np.size(np.where((decision_Q == 1) & (validation_class_labels == 1))))
TN_Q = (np.size(np.where((decision_Q == 0) & (validation_class_labels == 0))))
print("Error for Logistic Quadratic Model with ", train_data_sample_size, " samples is ", (10000 -
(TP_Q + TN_Q))/100)
plot_classified_data(validation_samples, validation_class_labels, decision_Q, optimal_solution_Q,
'Q')
```